DeVore, et al. A heterogeneous compute solution for
optimized genomic selection analysis.
Proceedings of the IEEE International
Conference on Bioinformatics and Biomedicine. IEEE, 2014.

A Heterogeneous Compute Solution for Optimized Genomic Selection Analysis

Trevor DeVore, Scott Winkleblack, Bruce Golden, Chris Lupo
California Polytechnic State University
San Luis Obispo, California 93407
{tdevore, swinkleb, bgolden, clupo}@calpoly.edu

***Abstract*—This paper presents a heterogeneous computing solution for an optimized genetic selection analysis tool, *GenSel*. GenSel can be used to efficiently infer the effects of genetic markers on a desired trait or to determine the genomic estimated breeding values (GEBV) of genotyped individuals. To predict which genetic markers are informational, GenSel performs Bayesian inference using Gibbs sampling, a Markov Chain Monte Carlo (MCMC) algorithm. Parallelizing this algorithm proves to be a technically challenging problem because there exists a loop carried dependence between each iteration of the Markov chain. The approach presented in this paper exploits both task-level parallelism (TLP) and data-level parallelism (DLP) that exists within each iteration of the Markov chain. More specifically, a combination of CPU threads using OpenMP and GPU threads using NVIDIA's CUDA paradigm is implemented to speed up the sampling of each genetic marker used in creating the model. Performance speedup will allow this algorithm to accommodate the expected increase in observations on animals and genetic markers per observation. The current implementation executes 1.84 times faster than the optimized CPU implementation.**

***Keywords*-Genetic Selection, GPU, Heterogeneous computing, Bayes methods, Monte Carlo methods, Markov processes**

I. Introduction

This paper presents a heterogeneous compute solution for the parallel optimization of GenSel, a genetic selection analysis tool which analyzes the genotype (genetic markers) and phenotype (physical traits) of an animal and attempts to isolate which of these markers contribute to a specific physical trait. This is achieved by applying Bayesian inference to the data set. This information can then be used, along with pedigree information, to increase the likelihood of breeding an animal with desirable traits. For example, breeding dairy cows with high milk production.

A. Motivation

The field of Bioinformatics relies heavily on statistical analysis of large quantities of data. Recently, the use of the Markov Chain Monte Carlo (MCMC) family of statistical algorithms has increased dramatically. As such, so has the desire to optimize these algorithms and increase their performance.

One challenge with performing this type of analysis is the quantity of data required to accurately predict what genetic markers influence a physical trait. Currently, a typical analysis by GenSel consists of approximately 50,000 genetic markers with 200,000 observations for each marker. Looking forward, the decreasing cost associated with performing genetic analysis is expected to increase the number of observations per marker to around 1,000,000 [1]. The current implementation of GenSel simply cannot scale to accommodate this. Optimizing the underlying MCMC algorithm of GenSel is a necessity.

MCMC algorithms strongly resist parallelization as a method for performance optimization. This class of algorithm does not fall into the embarrassingly parallel category and data dependence issues make it inherently iterative. However, if designed around parallel programming principles it is possible to achieve speedups of 10's to 100's of times [2]–[4] despite the iterative nature of the chain.

This paper takes a different approach which can be applied retroactively to MCMC based algorithms without requiring a drastic redesign of the original algorithm. This is achieved by exploiting smaller sections of parallelism that exist within each iteration of MCMC. Each section of potential parallelism that is identified is then mapped to the most beneficial parallel programming paradigm. The overall idea behind this approach is to reduce the runtime of every individual chain link, thereby reducing the time it takes for the entire chain to complete.

B. Contributions

The optimized implementation described in this paper utilizes heterogeneous compute methods to take advantage of CPU multi-threading, GPU computing via CUDA, utilizing multiple GPUs, pipelining, and single instruction multiple data (SIMD) instructions on the CPU. First, utilizing multiple CPU threads for the analysis improves the performance of course-grained task-level parallelism (TLP). Due to data dependencies that can occur between markers within one iteration of the chain, only a small number of loop tasks can be executed in parallel. This fits very nicely into the CPU threaded parallelization model. Second, the use of GPUs to tackle areas of data-level parallelism (DLP) is investigated. The Bayesian analysis used in GenSel requires several matrix operations, the size of which scales with the number of observations. This linear algebra is performed for every genetic marker in every iteration of MCMC. Performing this type of operation on the massively parallel GPU becomes beneficial for large matrices. For

smaller matrices where the communication overhead between the CPU and GPU is prohibitively expensive, the use of SIMD instructions to optimize matrix operations is explored. For increased parallelism and device utilization, kernel calls (function calls on the GPU) are performed concurrently when possible and computation is performed on multiple GPUs.

C. Outline

Section II presents background on the statistical analyses performed in GenSel, along with the algorithm being optimized. Related work is presented in Section III. The optimization methodology is described in Section IV. Details of the implementation are presented in Section V. The experimental setup is described in Section VI and results are presented in Section VII. Future work and conclusions are discussed in Sections VIII and IX, respectively.

II. Background

Bayesian inference is a form of statistical inference based around Bayes theorem, as the name suggests. In brief, Bayes theorem provides a methodology for updating beliefs based on evidence, taking into account the probability of not only the belief but also the evidence. The probability of the original belief is known as the prior and the probability of the belief given the evidence is known as the posterior.

This method of inference has found a wide variety of applications within the field of computer science. For example, Bayesian inference is used in artificial intelligence [5], [6] and spam filtering [7]. Recently, Bayesian inference has been increasingly leveraged by those studying phylogenetics [3].

This statistical technique is a powerful tool capable of delivering accurate estimates of genomic effects while remaining relatively straightforward to implement [1]. However, it is not without its challenges. Directly implementing Bayesian inference is problematic. For complex models there is no closed form solution for computing the posterior. Furthermore, the subset of solvable problems requires integrating over many dimensions, which can make the necessary computation infeasible.

One method of circumventing this problem, employed by GenSel, is to use a Gibbs sampler. Gibbs samplers belong to a class of algorithms known as Markov Chain Monte Carlo (MCMC) methods. They provide a practical way of generating empirical posterior distributions when the joint probability distribution of the variables of interest is difficult to sample directly.

In this application, each genetic marker can be viewed as a variable contributing to the desired trait. A statistical model is created that describes the mean effect of all markers on the desired trait. Another distribution captures an estimate of the error of the prediction. To construct these models, each marker is evaluated and is included or excluded from the model based on whether the marker meets or surpasses a threshold value. The threshold is based on a fully-conditional posterior variable [1]. This process is done repeatedly with the resulting model from iteration i forming the base model for iteration $i+1$, forming a chain. This algorithm represents the basis for the research described in this paper.

Gibbs sampling allows for genomic selection analyses to be performed in a reasonable compute time [1] but introduces problems of its own. To arrive at the correct posterior distribution using this technique requires repeated sampling until the probability distribution converges. Determining how many iterations it takes for a given distribution to converge proves to be a non-trivial problem [8]. Also, samples at the beginning of the chain are loosely correlated to the starting position, which is problematic. To overcome this, the concept of "burn in" or discarding the first portion of the chain is employed. MCMC algorithms are notorious for being compute intensive and any wasted computation only adds to the runtime.

MCMC algorithms are inherently serial in nature, making them difficult to parallelize for performance gains, although notable exceptions do exist [4]. The sequential nature of these algorithms is a consequence of the data dependence that occurs with the statistical model being produced in iteration i and consumed in $i+1$. Additionally, the `BayesC` algorithm presented in [1] contains an additional, conditional data dependence between markers when building the model in each chain link. This data dependence only occurs when a particular marker is selected to be included in the current model or excluded from the current model but was included in the previous model. This is a key observation that makes a subset of the optimizations presented in this paper possible.

III. Related Works

The emergence of MCMC algorithms into prominence coupled with its large computing cost has led to many attempts at optimization. These attempts can be broadly classified under two optimization strategies. One attempts to increase performance by running multiple MCMCs concurrently; the other seeks to parallelize a single MCMC run [9].

In the first strategy, multiple, shorter chains are run from different starting points and convergence is determined by comparing between and within sequence variance [10]. The performance gains come from the shorter chain lengths and the fact that each chain can be computed independently. The drawbacks of this approach are that the burn in period, which must be discarded, is repeated across multiple processes and for each iteration that is kept, after the burn in period, several more are discarded, a process known as thinning [11].

The second strategy recognizes the sequential, iterative nature of each chain link and attempts to optimize the operations that exist within each link. Consequently, by shrinking the amount of time spent in each link of the chain, the overall duration of the chain will be reduced even though it requires more

iterations. Within this strategy there are two sub-strategies associated with exploiting task-level or data-level parallelism.

The most successful approaches exploit data-level parallelism by utilizing GPU technology to parallelize general sampling methods [2] or the evaluation of likelihoods [3]. The results reported in research performing this type of optimization saw performance increases with factors of 10s to 100s [2]–[4].

Exploiting task-level parallelism turns out to be less successful than data-level parallelism in general. This is largely due to the fact that there is a dependence between each chain link which incurs major communication overhead due to the need for global model parameter updates [4]. Although some implementations have managed to achieve near linear speedup [12].

The implementation discussed in this paper adopts the second strategy of improving the performance of the overall chain by shrinking the size of the links.

Leveraging GPU technology as a complete solution, as in [2] or [4], proves to be problematic because of the additional data dependence between markers when building each model during each chain link. Additionally, this algorithm has problems with thread divergence (threads taking different execution paths), which does not map well into the GPU paradigm. Pre-computing the data responsible for control flow can solve the problems with divergence. But, pre-computing this data substantially reduces the amount of parallelizable code to the point that communication overhead between the host and device overshadows any performance improvement. Instead, use of GPU technology is limited to performing large matrix operations, which still provides a sizable speedup for very large matrices.

The implementation strategy in this paper is to exploit the varying levels of parallelism that exist within GenSel. By correctly identifying regions of code that exhibit parallelism and correctly associating these regions with the correct parallel programming paradigm based on system architecture considerations. This idea is set forth in [11] which mainly focuses more efficient matrix operations on the CPU. Our work expands the spectrum of parallelizable code and utilizes different technologies. This research illustrates that by exploiting several small performance gains it is possible to achieve a sizable performance improvement, without fundamental changes to the underlying MCMC algorithm.

IV. Methodology

Regions of code that are parallelizable to varying degrees are present throughout the GenSel application. The characteristics of each region including the system architecture, code divergence, the amount of data being processed, and runtime, determine which parallel programming paradigm is most appropriate for each parallelizable region. This section briefly describes the strategies taken with GenSel.

For the GenSel application, a heterogeneous computing environment was selected. The differing levels of parallelism offered by GPUs and CPUs make this environment well-suited to this problem decomposition. For the specifics on the hardware involved, refer to Section VI.

To exploit massive data-level parallelism, The GPU is utilized for linear algebra operations, including dot products of large vectors. Linear algebra tasks are high-performing on the GPU due to the amount of data processed, and the independence of each GPU thread.

To reduce the cost of transferring large vectors from the CPU to the GPU, the implementation presented in this paper utilizes a CUDA abstraction called a *stream* that allows for further exploitation of concurrency. Every kernel call is associated with a stream. A GPU device can have multiple streams associated with it at runtime. One major benefit of streams is, hardware permitting, they can be run concurrently. This achieves a pipelining effect where data transfers from the CPU to the GPU occur asynchronously and simultaneously with other computations on the GPU, minimizing the cost of the data transfer.

The ability to transfer data to and from the GPU simultaneously and asynchronously is important for performance. The largest bottleneck with GPU computing is often moving data between the CPU and GPU. By initiating data transfers as early as possible, data can be transferred while other work is being performed so that it is ready when needed.

CPU threads are better suited for independent or divergent execution than GPU threads. Exploiting parallelism on the CPU is most beneficial when the data transfer time to the GPU would be greater than the CPU compute time, or when the portion of code is highly divergent, or there is only a small amount of parallelism present.

Modern CPUs take advantage of data level parallelism via vectorized instructions. These instructions help to bridge the gap between GPU and CPU computing and make CPU computing more efficient than scalar processors for larger sets of data. The heterogeneous implementation described here utilizes all of these forms of parallelism and concurrency in appropriate regions of GenSel. Implementation details are discussed in the following section.

V. Implementation

This section presents the various optimizations performed on GenSel. Algorithm 1 gives an high level overview of the original algorithm.

A. Chunking Loops With Sporadic Dependencies

Due to the statistics behind the selection of which markers to include in the model, the loop dependency only occurs on average every 10 iterations. This means that while it is not

Algorithm 1 Original Bayes Algorithm

```
1: for each marker do
2:     compute probability marker is included in model
3:     if marker is included in model then
4:         compute mean genetic effect
5:         compute error prediction variance
6:         include marker in model
7:     else
8:         remove marker from model
9:     end if
10: end for
```

possible to parallelize the entire loop, it is possible to compute *chunks* of the loop at the same time to gain a performance advantage. Algorithm 2 shows a restructured Bayes algorithm that allows for increased parallelism in the computations.

Algorithm 2 New Bayes Algorithm

```
1: while markers processed < total number of markers do
2:     for each marker in chunk do
3:         compute probability marker is included in model
4:         if marker is to be included in model then
5:             set as terminating marker for chunk
6:         end if
7:     end for
8:     begin precomputing next chunk's dot products
9:     initiate observation data transfer to GPUs
10:    for each marker before terminal marker do
11:        if marker is to be included in model then
12:            compute mean effect
13:            compute error prediction variance
14:            include marker in model
15:        else
16:            remove marker from model
17:        end if
18:    end for
19:    update markers processed
20:    advance chunk to terminal marker
21: end while
```

In the first **for** loop, everything that is needed to determine which markers from the chunk are going to be included in the model is computed. The first marker that triggers the loop carried dependence is marked as the terminating marker of the chunk and the computed values of all following markers are discarded; they are no longer valid because they are based on the old model as opposed to the updated model. In the best case scenario, the entire chunk is valid and no computations are wasted, and in the worst case scenario, only one marker in the chunk is valid and the rest must be discarded. The optimal chunk size for a given system is determined partially on the statistics behind GenSel and partially on the system itself. The number of GPUs available is a large contributing factor in the latter. Properly selecting a chunk size results in upwards of 85% of the computations being valid. This keeps runtimes much closer to the best case than the worst case by keeping wasted computation down.

Next, the bottom **for** loop is executed for every marker in the chunk up to and including the terminating marker. This is responsible for updating the current model. There is no wasted computation associated with the second **for** loop. The GPU is not leveraged for this portion of the algorithm. The inability to predict which marker will be the terminal marker and transfer times associated with moving the necessary data to the GPU make the CPU better suited for the tasks. To increase efficiency of the matrix operations on the CPU, SIMD instructions are leveraged via the Eigen library [13].

Finally, the chunk is adjusted to the marker immediately following the terminating marker. The process is repeated until all the markers have been processed.

B. Algorithm Complexity Analysis

The computational complexity for Algorithm 2 is unchanged from that of Algorithm 1. For all computational complexity presented in this paper, n represents the input size in markers and c represents the chunk size, which is a constant defined by the user. The **for** loop in Algorithm 1 executes for each marker, and each operation in the loop is computed in constant time, and the algorithm is therefore $O(n)$ runtime complexity.

The analysis for Algorithm 2 is slightly more complex. The worst case occurs when every marker is included in the model for every iteration. In this case, the first marker in the chunk is the terminating marker but computation is performed on all other markers. This leads to wasted computation in the upper **for** loop. The second **for** loop only executes one iteration in this case. The chunk size c is much less than the number of markers n. This leads to a worst case computational complexity shown in Equation 1.

$$O(n(c+1)) \rightarrow O(n) \tag{1}$$

In the best case, the terminating marker in every chunk is the last marker. This occurs when no marker in the chunk causes the data dependence or the final marker causes the data dependence. In this case, the upper and lower **for** loops execute c times and enclosing while loop is executed $\lceil n/c \rceil$ times. This leads to a best case computational complexity shown in Equation 2.

$$O(\left\lceil \frac{n}{c} \right\rceil (c+c)) \rightarrow O(n) \tag{2}$$

The equivalent computational complexity between Algorithm 1 and Algorithm 2 indicates that any performance gains observed are not due to significant algorithmic refactoring.

C. Multi-Threading

Both **for** loops in Algorithm 2 lend themselves nicely to multi-threading. If the loop carried dependence is assumed to not exist for the markers concerned, they can be executed

independently. This assumption is checked before any changes are made to the model, as discussed in Subsection V-A. Since the number of markers for each chunk is known before execution, multi-threading principles are easy to apply. Each marker could be assigned to its own CPU thread. However, in practice one CPU thread managing each GPU is sufficient. When the number of observations grows large, the bottleneck becomes the computation performed by the GPU. Even with the GPU pipelining optimization, discussed in Subsection V-F, the CPU thread has sufficient time to perform all necessary computations before the next kernel call it is responsible for returns making multiple CPU threads per GPU unnecessary. This optimization allows GenSel to benefit from parallelization at both a data-level on the GPU and task-level on the CPU.

D. Parallelizing Dot Product Computation

The dot product computation performed to determine if a marker is to be included in the current model is the most time consuming portion of the algorithm. It is also one of the easiest aspects to optimize by leveraging parallelism. Computing the dot product of two vectors involves a large number of small operations with no dependencies. The technology of choice for this type of computation is the GPU. The large number of cores on a GPU allows for each of these smaller operations to occur simultaneously. By leveraging the cuBLAS linear algebra library [14], dot products can be performed on the GPU more quickly than on the CPU. Unfortunately, data must be transferred over the PCIe bus to the GPU before any computation can be done and back after it completes. This adds additional overhead and makes the cuBLAS dot product slower for smaller vectors. In this case, the size of the vector scales with the number of observations being considered, ensuring that the vectors will be sufficiently large. For the current implementation, leveraging the GPU was found to be advantageous when the number of observations was greater than 150,000.

E. Precomputing Dot Products

Computations on the GPU execute asynchronously from the CPU, allowing both to work at the same time. To take advantage of this feature, GPU computations should be launched as early as possible to maximize the amount of time the CPU and GPU are both doing work. In Algorithm 2, as soon as the first **for** loop terminates, all the information needed to compute the next chunk of dot products is readily available. Therefore, for efficiency, computation for the next chunk begins immediately. By overlapping CPU and GPU computation, the amount of time the CPU spends waiting for results is minimized.

F. Pipelining Kernel Calls

Making sure the GPU is saturated with work is extremely important for maximum efficiency. To ensure the GPU is saturated and executing as much work concurrently as possible, all dot product kernel calls for a given chunk are executed in separate streams. This signals to the GPU that the kernels are independent and may be run concurrently. Often it is not possible to run completely concurrently due to the limited number of streaming multi-processors on the device. As a result you get a pseudo pipelining effect. This helps to reduce the latency of each kernel call.

G. Multi-GPU and Striping

Launching a batch of dot products to the GPU scales very well when more GPUs are added. The current implementation is designed to scale seamlessly to accommodate an arbitrary number of GPUs. As the workload is spread across GPUs, more computations can be completed in a shorter amount of time, increasing throughput and decreasing the amount of time the CPU must wait for the dot product results. Dividing the marker index by the number of GPUs and using the remainder as an index into the set of GPUs provides a simple and effective way of load balancing calls.

Another benefit of utilizing multiple GPUs is data striping. For dot product computations, each GPU only needs to be aware of the observation vectors associated with the markers it is being asked to perform computations for. The load balancing scheme defined above makes it easy to know which data is to be associated with which GPU. Therefore, the total amount of data transferred to all the GPUs remains constant. However, if the GPUs are located on different PCIe buses there is additional data transfer bandwidth, leading to increased performance.

H. Circular Buffering of Data on the GPU

One of the biggest drawbacks to GPU computing is the amount of time it takes to transfer data across the PCIe bus. Optimally, all of the data required for all computations would already be on the GPU by the time computation begins. Unfortunately, the comparatively large size of the observation vectors and small size of GPU memory prohibits this, even when data is striped across multiple GPUs. But, for the data sizes within the scope of this paper, all data associated with the next n chunks can be stored on the GPU. N is a configurable value, allowing customization to the system in use. N should be set large enough so the GPU does not have to wait to begin computation but small enough to fit all the data on the GPU. The n value used throughout this paper is 4, and was determined experimentally to sufficiently prevent GPU starvation.

The portion of the data associated with the next n chunks needed by each GPU resides in a circular buffer on the respective GPU. When a dot product is computed and is not discarded, the data vector is replaced by the next vector. This transfer is initiated by line 9 of Algorithm 2.

Holding the next several chunks worth of data on the GPU has many benefits. First, it allows the current chunk's worth

of dot products to be computed uninterrupted. Also, it allows computation on the next chunk's markers to begin immediately without waiting for data to be transferred over.

I. Hiding Memory Transfers

In most applications that utilize the GPU, a large portion of time is spent transferring memory back and forth between the CPU and GPU. There are several known techniques for amortizing this cost. One option is using asynchronous memory transfers. These types of transfers return control to the CPU immediately, instead of waiting for the transfer to complete. This allows the CPU to do additional work while the memory is transferring to the device. The GPU can also be performing work while data is being transferred. Additionally, all GPUs used in this paper have two copy engines, enabling them to transfer data to and from the CPU simultaneously.

All memory transfers occur asynchronously to overlap data transfer with execution as much as possible. To minimize or eliminate the amount of time that the CPU is waiting on the GPU or vice versa, they are performed as early as possible. This optimization operating in conjunction with the data buffering technique discussed in Subsection V-H effectively eliminate the time the GPU is waiting for data to perform its calculations. The initial transfer of data to the GPU is done during the setup phase of the GenSel application. All subsequent transfers occur while the CPU is updating the current model and the GPU is precomputing the next chunk's dot products.

VI. Experimental Setup

A. Test System

All performance measurements were performed on a single system. The system has two Intel Xeon E5-2650 8-core CPUs which run at 2.00 GHz and support hyper-threading. The system has 64 GB of RAM. When generating results, three GPUs were utilized; one NVIDIA Tesla K40 and two NVIDIA K20Xs. The Tesla K40 has 2880 cores and 12 GB of memory while the K20Xs have 2688 cores and 6 GB of memory. They run at 745 MHz and 732 MHz respectively.

The following information describes the software environment these experiments were conducted in. All source code was compiled using GCC version 4.8.2 and CUDA 6.0, when applicable. The NVIDIA driver version is 334.21. The system was running Arch Linux with version 3.14.0 of the Linux kernel.

B. Test Data

The goal of these optimizations is to help GenSel scale to accommodate the rising amount of genotype information available in the future. Consequently, to perform relevant tests it was necessary to generate simulated genotypic and phenotypic data. For this purpose, a data generating program was created that can create data sets with an arbitrary number of markers and observations per marker. To make run times reasonable for experimentation, data sets were limited to 10,000 markers.

C. Reference Implementation and Validation

The baseline CPU implementation makes use of the highly optimized Eigen library [13]. The Eigen library leverages vectorized instructions, loop unrolling, and cache optimizations.

The baseline implementation was run on all of the generated test data sets. The results of the analyses were compared against the optimized heterogeneous computing solution to confirm correctness. In addition, timing data was recorded to be used as a point of comparison for benchmarking. The timing data reported includes compute time (including data transfer times) for the optimized baseline and heterogeneous solution, system I/O time (e.g., disk) is excluded.

VII. Results

This section presents the results of all the optimizations performed on the heterogeneous compute solution. All results presented represent the speedup relative to the highly optimized baseline CPU implementation. The measured runtimes do not include IO operations, such as reading in the genotypic and phenotypic data from disk, which is common to both implementations. The runtimes include all computation performed for the analysis including all bus transfers between the CPU and GPU. When generating the runtimes all three GPUs of the system were utilized, unless otherwise stated. The chunk size was set to six markers concurrently and four chunks worth of data was held in each GPU's buffer.

Fig. 1 presents the overall speedup of the heterogeneous compute solution for varying amounts of markers. As mentioned earlier, the number of markers was limited to 10,000 for the experiments. In the best case for these data sets a 1.84 times speedup is achieved over the optimized baseline CPU implementation. These results demonstrate that the cutoff point at which it becomes beneficial to use the GPU over the CPU is about 150,000 markers. At that point the massive parallelism offered by the GPU outstrips the CPUs ability to perform large matrix operations. This cutoff is not constant. It varies based on the number of GPUs used and the chunk size, as shown in Figs. 2 and 3.

The tight clustering of the speedup data illustrates that the optimizations performed are not proportional to the number of markers and will scale to accommodate higher marker counts. The primary factor governing how large the speedup can be is the number of observations. This largely determines the speedup of the calculations for any given marker because the larger data size favors the GPU approach leveraged by the optimized heterogeneous solution over the optimized pure CPU approach. However, since all calculations for all markers cannot be performed in parallel, performance gains due

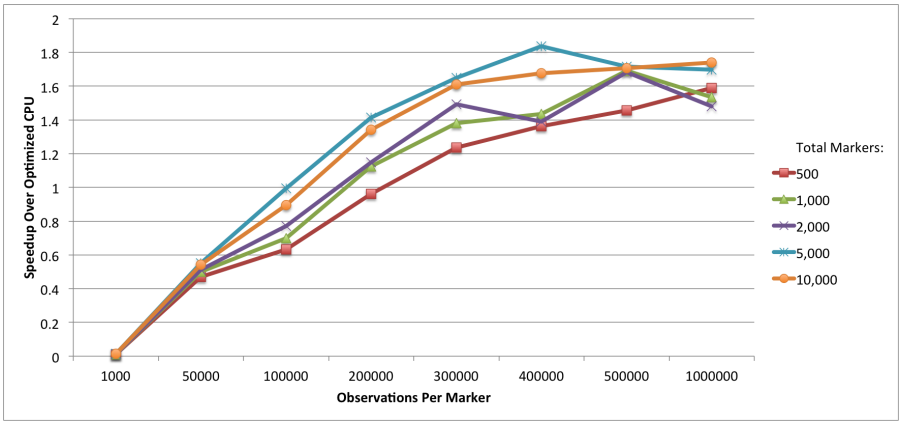

Fig. 1. Overall speedup with varied number of markers

TABLE I
RUNTIMES FOR 500 MARKER DATA SET

Number of Observations	CPU Runtime (s)	GPU Runtime (s)	Speedup
1,000	14	1,284	0.01
50,000	797	1,692	0.47
100,000	1,579	2,493	0.63
200,000	3,184	3,316	0.96
300,000	4,761	3,855	1.24
400,000	6,375	4,678	1.36
500,000	7,920	5,435	1.46
1,000,000	15,653	9,850	1.59

to coarse-grained parallelism are limited to the chunk size. Chunking the calculations reduces the number of iterations required to build the statistical model for one link in the MCMC chain. The overall speedup offered by the heterogeneous solution is the product of the per marker speedup times the amount of parallelism offered by the chunk size. The amount of parallelism offered by chunking can be thought of as a fixed quantity due to the statistical properties of the problem. The per marker speedup does not vary by the number of markers because it is based on the number of observations. Thus, the speedup remains relatively constant as the number of markers scales.

Table I shows the runtimes for the 500 marker case. The runtimes scale as expected with respect to marker growth, meaning the runtimes for the 1000 marker case are about two times larger than the runtimes in Table I.

A. Tuning

Tuning a high performance application to its intended computing environment is critical and can have dramatic effects on the performance of the application. In this case, there are two main values that must be considered: the size of the per GPU buffer and the chunk size. The GPU buffer size can easily be statically determined by examining the amount of memory available on all the GPUs. The current implementation creates the same buffer size on all GPUs for simplicity. So, the smallest memory size of the GPUs the user wishes to use

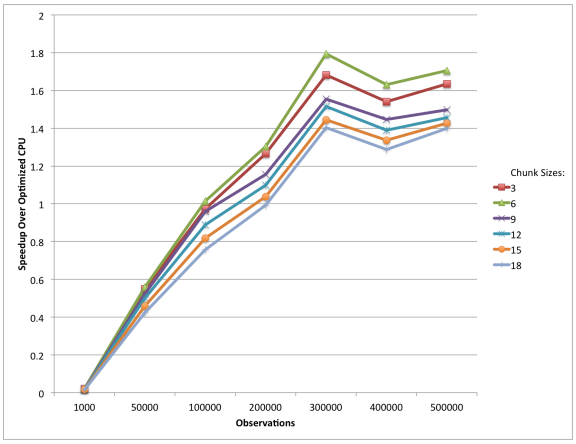

Fig. 2. Effect of different chunk sizes on speedup

becomes the limiting factor. For the GPUs in this work, 6 GB is the smallest global memory size.

Tuning the chunk size is much more interesting due to the many different contributing factors. For optimal performance, several different aspects of the system must be considered: the number of threads available on the system, the number of GPUs connected, and if the GPUs share a common bus. The other major contributing factor to tuning the chunk size is the statistics behind GenSel. On average any given marker is excluded with a probability of .95. Thus, for any given marker the probability that it was included in the current iteration or excluded but was included in the iteration immediately previous is .90. Meaning, on average a data dependence will occur one out of every ten markers. The optimal chunk size is easily determined empirically. Fig. 2 demonstrates the effect of different chunk sizes on the speedup of the application for the 500 marker data set.

B. Scalability

The heterogeneous solution is designed to effortlessly scale to leverage additional GPUs. It can be easily configured to

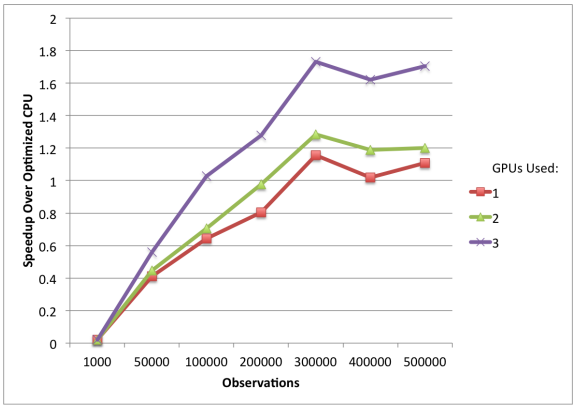

Fig. 3. Effect of scaling the number of GPUs utilized on speedup

leverage all or some of the GPUs available on the system. GPUs can also be assigned priority giving the user control over which GPUs are selected for use if they are not all utilized.

Fig. 3 illustrates the speedup as more devices are made visible to the program. As mentioned in Section VI, the system used for generating these results has one NVIDIA K40 and two K20X GPUs available for use. When generating these results the K40 was always used and the K20X GPUs were added as the number of GPUs was scaled. The results presented are for the 500 marker data set.

All of the results show a leveling-off of performance compared to the baseline CPU implementation for large numbers of observations. While the GPUs allow for massively parallel computing, the amount of parallelism is finite. Once the GPUs become saturated, no additional performance increase is expected. As the number of cores per GPU increases, further speedup is expected at larger numbers of observations.

VIII. Future Work

This paper focuses on optimizing GenSel for large data sets both in terms of markers and observations. The optimizations presented in this paper come at the expense of sacrificing performance for smaller data sets. In the future, a heuristic based on the number of observations and GPUs available could be used to quickly select between implementations to dynamically choose the best computational method.

GenSel has other opportunities for optimization. Those covered in this research have avoided fundamentally changing the underlying algorithm. However, one of the most promising optimizations requires a shift to another form of Gibbs sampling. Changing to block Gibbs sampling may enable processing of larger subsections of markers than the current chunking method.

IX. Conclusion

The process of optimizing genetic selection analysis for heterogeneous computing is explored. This research shows that by identifying small regions of TLP and DLP that exist within largely sequential algorithms, such as MCMC, the accumulated speedup can be significant, without fundamentally changing the underlying MCMC algorithm. The optimizations presented in this paper achieve a 1.84 times speedup. This research will enable GenSel to scale in the future, and to benefit from increasing hardware parallelism on both the CPU and GPU.

Acknowledgments

This research is/was supported by the USDA National Institute of Food and Agriculture, AFRI competitive grants program project 0227921.

References

[1] D. Habier, R. L. Fernando, K. Kizilkaya, and D. J. Garrick, "Extension of the Bayesian alphabet for genomic selection," *BMC bioinformatics*, vol. 12, no. 1, p. 186, 2011.

[2] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes, "On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods," *Journal of Computational and Graphical Statistics*, vol. 19, no. 4, pp. 769–789, 2010.

[3] M. A. Suchard and A. Rambaut, "Many-core algorithms for statistical phylogenetics," *Bioinformatics*, vol. 25, no. 11, pp. 1370–1376, 2009.

[4] M. A. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, "Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures," *Journal of Computational and Graphical Statistics*, vol. 19, no. 2, 2010.

[5] C. Huang and A. Darwiche, "Inference in belief networks: A procedural guide," *International Journal of Approximate Reasoning*, vol. 15, no. 3, pp. 225–263, 1996.

[6] G. F. Cooper, "The computational complexity of probabilistic inference using Bayesian belief networks," *Artificial intelligence*, vol. 42, no. 2, pp. 393–405, 1990.

[7] G. V. Cormack and T. R. Lynam, "Online supervised spam filter evaluation," *ACM Transactions on Information Systems (TOIS)*, vol. 25, no. 3, p. 11, 2007.

[8] A. E. Raftery, S. Lewis *et al.*, "How many iterations in the Gibbs sampler," *Bayesian statistics*, vol. 4, no. 2, pp. 763–773, 1992.

[9] S. Brooks, "Markov chain Monte Carlo method and its application," *Journal of the royal statistical society: series D (the Statistician)*, vol. 47, no. 1, pp. 69–100, 1998.

[10] A. A. Béguin and C. A. Glas, "MCMC estimation and some model-fit analysis of multidimensional IRT models," *Psychometrika*, vol. 66, no. 4, pp. 541–561, 2001.

[11] J. E. Gentle, W. Härdle, and Y. Mori, *Handbook of computational statistics*. Springer, 2004.

[12] F. Ronquist and J. P. Huelsenbeck, "MrBayes 3: Bayesian phylogenetic inference under mixed models," *Bioinformatics*, vol. 19, no. 12, pp. 1572–1574, 2003.

[13] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[14] NVIDIA Corporation, "CUDA toolkit documentation – cuBLAS," http://docs.nvidia.com/cuda/cublas/, February 2014.